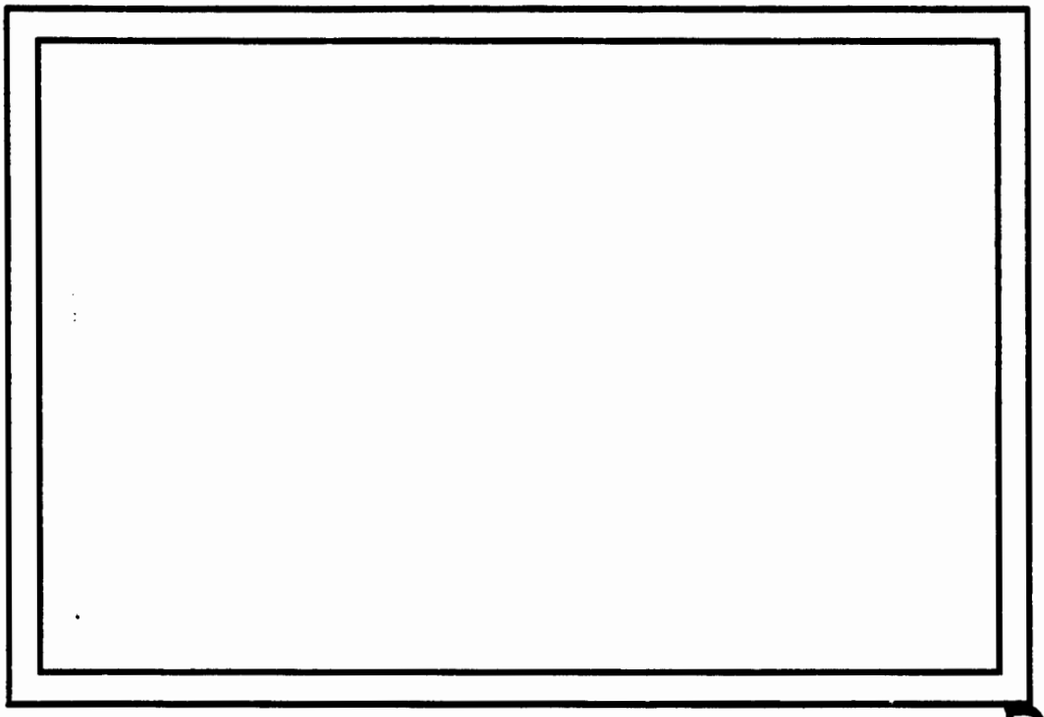


UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D
(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author) Computer Science Center University of Maryland College Park, Md. 20742	2a REPORT SECURITY CLASSIFICATION Unclassified 2b GROUP -

3 REPORT TITLE

Array automata and array grammars

4 DESCRIPTIVE NOTES (Type of report and inclusive dates)

Technical Report

5 AUTHOR(S) (Last name, first name, initial)

Milgram, David L.
Rosenfeld, Azriel

6 REPORT DATE	7a TOTAL NO OF PAGES	7b NO OF REFS
November 1970	24	5

8a CONTRACT OR GRANT NO Nonr-5144(00) b PROJECT NO c d	9a ORIGINATOR'S REPORT NUMBER(S) TR-70-141 9b OTHER REPORT NO(S) (Any other numbers that may be assigned this report) -

10 AVAILABILITY/LIMITATION NOTICES

11 SUPPLEMENTARY NOTES	12 SPONSORING MILITARY ACTIVITY Information Systems Branch Office of Naval Research Washington, D. C.

13 ABSTRACT

It is shown that grammars that rewrite arrays are equivalent to Turing machines having array "tapes", and that "monotonic" array grammars (in which arrays never shrink in the course of a derivation) are equivalent to "array-bounded" machines. It is also shown that two alternative definitions of "array-bounded" are in fact equivalent.

DD FORM 1 JAN 64 1473

UNCLASSIFIED
Security Classification

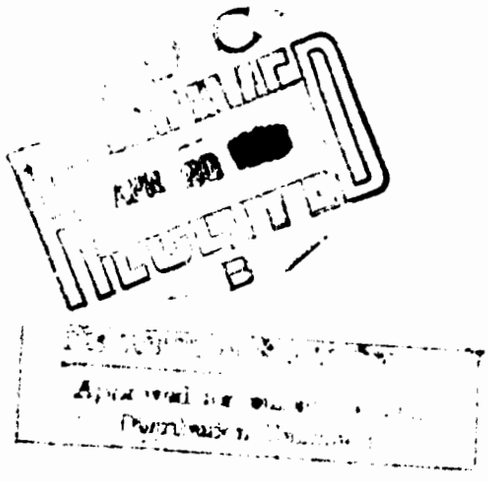
Technical Report 70-141 November 1970
Nonr-5144(00)

ARRAY AUTOMATA AND ARRAY GRAMMARS

David L. Milgram
and
Azriel Rosenfeld

ABSTRACT

It is shown that grammars that rewrite arrays are equivalent to Turing machines having array "tapes", and that "monotonic" array grammars (in which arrays never shrink in the course of a derivation) are equivalent to "array-bounded" machines. It is also shown that two alternative definitions of "array-bounded" are in fact equivalent.



RESEARCH PROGRESS REPORT

TITLE: Array automata and array grammars, David L. Milgram and A. Rosenfeld, University of Maryland Computer Science Center Technical Report 70-141, November 1970; Contract Nonr-5144(00).

BACKGROUND: The Computer Science Center of the University of Maryland is investigating the theory of image processing by computer, with emphasis on the theory of "picture grammars".

CONDENSED REPORT CONTENTS: It is shown that grammars that rewrite arrays are equivalent to Turing machines having array "tapes", and that "monotonic" array grammars (in which arrays never shrink in the course of a derivation) are equivalent to "array-bounded" machines. It is also shown that two alternative definitions of "array-bounded" are in fact equivalent.

FOR FURTHER INFORMATION: The complete report is available in the major Navy technical libraries and can be obtained from the Defense Documentation Center. A few copies are available for distribution by the author.

1. Introduction

During the past few years, several investigators have studied automata that make use of two-dimensional (or even n-dimensional) "tapes" [1-3]. There has also been some interest in grammars whose languages are sets of arrays rather than sets of strings [4-5].

The purpose of this paper is to show that array grammars are equivalent to Turing machines with array "tapes", and that "monotonic" array grammars (in which arrays never shrink in the course of a derivation) are equivalent to "array-bounded" machines (analogous to linear-bounded automata). It is also shown that two alternative definitions of "array-bounded" are in fact equivalent.

2. Array grammars and Turing machines

We define a Turing array acceptor (TAA) to be a 5-tuple $T = (Q,\Sigma,\delta,q_s,F)$, where

Q is a finite set of states of the form $Q' \times \Delta$ $(\Delta = \{L,R,U,D\})$

Σ is a finite set of symbols

$(q_s,R) \in Q$ is a start state

$F \subseteq Q$ is a set of final states

δ is a mapping from $Q \times \Sigma$ into $2^{Q \times \Sigma \times \Delta}$ such that the triples in the image sets are all of the form $((q,Y),B,Y)$, $Y \in \Delta$.

Here Q' can be thought of as a set of "internal" states and Δ as a set of directions ("left", "right", "up", "down"); for reasons that will become apparent below, it is convenient if the state of T at any time contains information about the direction that T has just come from. The interpretation of δ is as follows: If T is in internal state p and has just moved in direction X, and it reads symbol A, it goes into some internal state q, writes symbol B, and moves in direction Y.

T is called deterministic if the image under δ of every pair in $Q \times \Sigma$ is a singleton;* otherwise, nondeterministic. T is called finite-state (an FSAA, for short) if it can never rewrite the symbols that it reads -- in other words, if every triple in the image of any $((p,X),A)$ under δ is of the form $((q,Y),A,Y)$.

Let σ be an array on Σ, i.e. a mapping from $I \times I$ (where I is the integers) into Σ. The image of $(i,j) \in I \times I$ under σ

---

* For brevity, we shall omit the braces and represent these singletons by their sole elements.

will be called the value of (i,j). We want to define a notion of "acceptance" of G by T along the following lines: T starts at some point of IxI in its start state, and reads (and rewrites) the values of points as it moves around; it "accepts" G if it ever goes into a final state. Clearly, however, if we do not somehow restrict G to some finite portion of IxI, and require T's starting point to lie in that portion, we cannot guarantee that T will be able to see all of G in any finite number of moves, so that acceptance will have to be based on incomplete information. We shall therefore assume here that Z contains a distinguished symbol #; that in any array, all but finitely many points have value #; and that when given an array to test for acceptance, T starts at a point that does not have value #, if such a point exists. Moreover, we shall assume that the set P of points that have values other than # is connected*; if we did not do so, it is evident that T could never know when it has seen all of P.

In summary: By an input array on Z we mean a mapping G from IxI into Z such that the preimage P of Z-{#} is finite and connected. We allow T to operate on G by starting with a pair whose terms are the start state of T and a point (i,j) of P (the initial "position" of T). The mapping δ is applied to the pair $((q_s,R),A)$, where A is the value of G at (i,j).

---

* We say that P is connected if for any (i,j), (h,k) in P there exist $(i_0,j_0),\ldots,(i_n,j_n)$ in P such that $(i_0,j_0) = (i,j)$, $(i_n,j_n) = (h,k)$, and $|i_m-i_{m-1}| + |j_m-j_{m-1}| = 1$, $1 \leq m \leq n$ ("(i_m,j_m) and (i_{m-1},j_{m-1}) are adjacent").

If $((q,X),B,X)$ is a triple in $\delta((q_s,R),A)$, we regard T as having rewritten A as B (i.e., G now has value B at (i,j)), and we regard the new position of T as being

$(i-1,j)$, if X=L $(i+1,j)$, if X=R

$(i,j+1)$, if X=U $(i,j-1)$, if X=D

We can then apply the mapping δ to the pair $((q,X),C)$, where C is the value of G at the new position. If any such sequence of applications of δ leads to a triple whose first term is in F, we say that T <u>accepts</u> G.

We define an <u>array grammar</u> (AG) to be a 5-tuple $G = (V,\Sigma,R,\#,S)$, where

V is a finite set of <u>symbols</u>, called the <u>vocabulary</u>

$\Sigma \subset V$ is called the <u>terminal vocabulary</u>

$\# \in V-\Sigma$ is called the <u>blank symbol</u>

$S \in V-\Sigma$ is called the <u>initial symbol</u>

R is a finite set of <u>rewriting rules</u>, each of which is a pair (G_P, G'_P), where P is a finite connected subset of IxI, and G_P, G'_P are mappings from P into V. We assume that terminal symbols are never rewritten, i.e. that if $(i,j)\in P$ and G_P takes (i,j) into $a\in\Sigma$, then G'_P also takes (i,j) into a.

We say that the array $\mathcal{B}'$ on V is directly derivable in G from the array $\mathcal{B}$ if, for some (G_P, G'_P) in R, there exists a translate P* of P such that the restriction of $\mathcal{B}$ to P* is G_P, and $\mathcal{B}'$ is the same as $\mathcal{B}$ off P* and is equal to G'_P on P*. We say that $\mathcal{B}'$ is derivable in G from $\mathcal{B}$ if there exist $\mathcal{B} = \mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_n = \mathcal{B}'$ such that $\mathcal{B}_m$ is directly derivable from $\mathcal{B}_{m-1}$, $1\leq m\leq n$. By an

initial array we mean an array on {S,#} in which the preimage of S consists of exactly one point; by a terminal array we mean an array on $\Sigma \cup \{\#\}$ in which the preimage of Σ is connected. By the language of G we mean the set of terminal arrays that are derivable from initial arrays. (Evidently, in any such terminal array the preimage of Σ must be finite, since the array results from a finite number of rewriting rule applications, each of which can create only finitely many symbols in Σ.)

It is easy to show, just as for string grammars, that if G is any AG, there exists an AG G', having the same language as G, whose rules all have P's consisting of a single point (i,j) or a pair of adjacent points; it can also be assumed, in the latter case, that G_P and G'_P both give (i,j) the same value. We shall use the notation $A \to B$ for a rule in which P is a single point, and the values given to P by the two members of the rule are A and B, respectively. Similarly, let P consist of two points, one of which (say (i,j)) is given the values C and C' by the two members of the rule, while the other point gets the respective values A and B.

If the other point is	We denote the rule by
(i-1,j)	$CA \to C'B$
(i+1,j)	$AC \to BC'$
(i,j-1)	$\begin{matrix} C \\ A \end{matrix} \to \begin{matrix} C' \\ B \end{matrix}$
(i,j+1)	$\begin{matrix} A \\ C \end{matrix} \to \begin{matrix} B \\ C' \end{matrix}$

It will sometimes be convenient to denote CA by $C_R A$, AC by $C_L A$, $\begin{matrix} C \\ A \end{matrix}$ by $C_D A$, and $\begin{matrix} A \\ C \end{matrix}$ by $C_U A$; a generic two-point rule member can then be denoted by $C_X A$ (X = L,R,D, or U).

Our goal in this section is to establish

Theorem 1. Let $\mathfrak{L}$ be the language of an AG; then there exists a TAA that accepts just the arrays of $\mathfrak{L}$. Conversely, let $\mathfrak{S}$ be the set of input arrays accepted by a TAA; then there exists an AG whose language is $\mathfrak{S}$.

We first need

Proposition 2. Let Σ' be a finite set of symbols, and suppose that Σ' is the disjoint union of Σ_1, Σ_2, and $\{\#\}$. There exists a deterministic TAA which, given an input array on Σ' and an initial position in the preimage of $\Sigma_1 \cup \Sigma_2$, will accept the array if and only if the preimage of Σ_2 is empty.

Proof: One can evidently define a deterministic TAA that follows a space-filling "square spiral" around its initial position, marking the tape (by rewriting its symbols) as it goes. If it finds a symbol in Σ_2, it goes into an absorbing non-final state; if it has not yet found such a symbol, and finds nothing but #'s for a complete turn of the spiral, it goes into a final state. Since the preimage of $\Sigma_1 \cup \Sigma_2$ on an input array must be finite and connected, it is clear that this TAA will go into its final state if and only if the preimage of Σ_2 is empty. Note that if the array contains nothing but #'s, our TAA can accept it immediately, since by our conventions, only then can its initial position be on a #. //

It follows readily from the proof of Proposition 2 that a deterministic TAA can always find (e.g.) the "upper left hand corner" of its input array -- i.e., the leftmost non-# on the

highest row that contains non-#'s. Thus there is no need to require, in defining the notion of acceptance, that the TAA always start in a distinguished position on the array.

Using the proof of Proposition 2, one could also describe a deterministic TAA that circumscribes a rectangle (say m by n) around the non-#'s in its input array, and then maps the rectangle into a string of length mn. It could also create an extra segment of length m to use as a counter in making moves of exactly m steps to the right or left on the string, which simulate single moves up or down on the original array; thus it could imitate a given TAA while moving only on a string. Whenever the given TAA had to extend the array, its string simulator could lengthen the string appropriately. Since in the string case, nondeterministic Turing acceptors are equivalent to deterministic ones, this construction can be used to establish the analogous result for the array case. We could also use this approach to prove Theorem 1, but shall give a direct proof instead.

To prove the first part of Theorem 1, let

$Q' = \{q_s, q_t\} \cup \{q_A \mid A \in V\}$, where V is the vocabulary of the given AG

$\Sigma = V' \cup (V' \times V)$, where V' consists of the terminal vocabulary of the given AG together with #

For all $a \in V'$, let

1) $\delta((q_s, X), a) = \{((q_s, Y), (a, \#), Y) \mid Y \in \Delta\} \cup \{((q_s, Y), (a, S), Y) \mid Y \in \Delta\}$ for all $X \in \Delta$, where S is the initial symbol of the given AG. In other words, our TAA can remain in the start internal

state and move around, rewriting a's as (a,#)'s; it changes from this state only when it rewrites one of the a's as (a,S).

2) $\delta((q_C,X),a) = T_{a\#} \cup T'_{a\#} \cup T_{aCX\#}$

$\delta((q_C,X),(a,A)) = T_{aA} \cup T'_{aA} \cup T_{aCXA} \cup T_{XaA}$ for all $A \in V$, $C \in V$ and $X \in \Delta$, where

$T_{aA} = \{((q_A,Y),(a,A),Y) \mid Y \in \Delta\}$

In other words, when our TAA is in one of the q_C states, it can also more around arbitrarily, rewriting a's as (a,#)'s if it finds them; at each move, its internal state changes to match the second term of the symbol-pair it has just left.

$T'_{aA} = \{((q_B,Y),(a,B),Y) \mid Y \in \Delta;$ $A \to B$ a rule of the given AG$\}$

Alternatively, if our TAA is in a q_C state and finds a symbol A, or second term A, such that $A \to B$ is a rule, it can apply the rule.

$T_{aCXA} = \{((q_B,Y),(a,B),Y) \mid Y \in \Delta;$ $C_XA \to C_XB$ a rule of the AG$\}$

Similarly, if our TAA is in state (q_C,X), and finds a symbol A or second term A such that $C_XA \to C_XB$ is a rule, it can apply the rule. By our earlier remarks, we can assume that all the rules of the given AG are of these forms. Thus our TAA can move around and imitate, on second terms of pairs, the action of the AG's rules; the first terms of pairs remain unchanged.

$T_{XaA} = \{((q_t,X^{-1}),(a,A),X^{-1})\}$, where $L^{-1}=R$, $R^{-1}=L$, $U^{-1}=D$, $D^{-1}=U$: At any stage, our TAA can also go into the new internal state q_t and reverse its latest move (so that it always moves back to a pair).

3) Throughout (1-2), the set of points having values other than # always remains connected. We thus know that, using Proposition 2, we can now introduce new states and symbols, and extend the definition of δ, so that when our TAA has gone into the q_t state, it deterministically tests the array to check whether every non-symbol is a pair of the form (a,a) for some $a \in V'$. Evidently, this can be the case if and only if the original array was in the language of the given AG. If so, the TAA enters a final state; otherwise not. Note that even if the original array consisted entirely of #'s, the array tested always has pairs in it, so that its non-# part is nonempty.

To prove the second part of the theorem, let

$$V = \Sigma \cup (\Sigma \times \Sigma \times (Q \cup \{0,1\}))$$

where Σ is the set of symbols and Q the set of states of the given TAA. Let $\Sigma - \{\#\} \subset V$ be the terminal vocabulary; let $(\#,\#,1) \in V$ be the initial symbol; and let R consist of the following rules:

1) $(\#,\#,1) \rightarrow (a,a,1)$

$(a,a,1)_X \# \rightarrow (a,a,0)_X (b,b,1)$

$(a,a,1)_X (b,b,0) \rightarrow (a,a,0)_X (b,b,1)$

for all a,b in $\Sigma - \{\#\}$ and all $X \in \Delta$. These rules create, from an initial array consisting entirely of #'s except for a single (#,#,1), an arbitrary finite connected array of (a,a,0)'s, in just one of which the third term is 1 rather than 0.

2) $(a,a,1) \rightarrow (a,a,(q_s,R))$ for all $a \in \Sigma - \{\#\}$

At any stage of (1), the (a,a,1) can turn into a triple whose third term is the start state of the given TAA.

3) $(a,b,(p,X))_Y(c,d,0) \rightarrow (a,e,0)_Y(c,d,(q,Y))$ for all a,c,d in Σ;

$(a,b,(p,X))_Y\# \rightarrow (a,e,0)_Y(\#,\#,(q,Y))$ for all $a \in \Sigma$

-- provided $((q,Y),e,Y) \in \delta((p,X),b)$ for the given TAA and (p,X) is not a final state. Using these rules, the grammar now simulates the behavior of the TAA on the input array G of second terms of the triples created by (1). The unique point with nonzero third term represents the TAA's position at any stage.

4) $(a,b,(p,X)) \rightarrow a$ for all a,b in Σ and all final states (p,X);

$a_X(b,c,0) \rightarrow a_X b$ for all a,b,c in Σ and all $X \in \Delta$.

If the simulated TAA enters a final state, the array of triples can be turned into the array G. Thus G is a terminal array of our grammar if and only if it is an array that the given TAA accepts. //

3. Monotonic array grammars and array-bounded machines

A TAA will be called array-bounded (an ABA) if it "bounces off" #'s, i.e., if every triple in the image of any $((p,X),\#)$ under δ is of the form $((q,X^{-1}),\#,X^{-1})$. It will be assumed that the input array of an ABA always contains a non-#.

An AG will be called monotonic (an MAG) if it cannot create #'s -- in other words, if (G_p, G'_p) is any rule, and G'_p takes $(i,j)\varepsilon P$ into #, so does G_p; thus in the course of any derivation, the number of non-#'s is monotonically nondecreasing. Our goal in this section is to prove

Theorem 3. Let $\mathfrak{L}$ be the language of an MAG; then there exists an ABA that accepts just the arrays of $\mathfrak{L}$. Conversely, let $\mathfrak{S}$ be the set of input arrays accepted by an ABA; then there exists an MAG whose language is $\mathfrak{S}$.

Our proof will make use of the analog of Proposition 2 for ABA's (see Theorem 4 below); given this, Theorem 3 follows readily from the proof of Theorem 1.

To see the first part, note that in the proof of the first part of Theorem 1, if the AG is monotonic and generates the given input array, it must do so from an initial S located at one of the non-#'s of that array, since otherwise it would have to create at least one # (where the S initially was). Thus in step (1) of the proof, our TAA need not leave the non-#'s. Similarly, in step (2), the symbols rewritten by the rules of the AG need never lie outside the non-# points, so that our TAA need never leave these points to apply these rules. (It can, however, detect #'s that are adjacent to non-#'s when it

"bounces off" them, so that it can use them as context when necessary.) Finally, in step (3), the analog of Proposition 2 can be used to show than an ABA can check, without leaving the non-#'s, whether they are all pairs of the form (a,a).

Conversely, in the second part of the proof of Theorem 1, the only rules that could possibly create #'s are those in (4); but if the TAA being simulated bounces off #'s, we need never turn them into triples, so that the rules in (4) will never create #'s. //

One could also consider array grammars that are "isotonic" in the sense that #'s are neither created nor destroyed by any rule, so that the number of non-#'s remains constant throughout any derivation: here initial arrays having arbitrary finite connected sets of S's, rather than just a single S, would have to be allowed (see [5]). It is clear that if one of these S's is distinguished from the others (call the others T's), such grammars are exactly as powerful as MAG's, since we can regard the T's as the #'s that a MAG would have destroyed (and never re-created) in the course of a derivation. A more difficult question is whether such grammars are still as powerful as MAG's if there is no distinguished S. (This question does not arise for strings, since (e.g.) the leftmost S is always distinguished by virtue of having a # to its left, and it can then be shifted into any position. Similarly, if arrays of non-#'s are always assumed to be rectangular, there are always distinguished S's, e.g. those in the corners.)

Theorem 4. Let Σ' be as in Proposition 2. There exists a deterministic ABA which, given an input array on Σ' and an initial position in the preimage of $\Sigma_1 \cup \Sigma_2$, will accept the array if and only if the preimage of Σ_2 is empty.

Proof: By our definition of ABA, the preimage P of $\Sigma_1 \cup \Sigma_2$ is nonempty. Some of the points of P are boundary points, i.e., they are adjacent to #'s; some of these boundary points belong to the "outer boundary" of P, while others may belong to the boundaries of "holes" in P (i.e., connected components of #'s that are completely surrounded by P). Each of these boundaries can be regarded as the discrete analog of a simple closed curve, and it is not difficult to define a deterministic ABA that can "follow" any such boundary in a specified sense (say counterclockwise) without being confused by the fact that the boundary may share points with other boundaries, or may even pass through some points more than once.

We shall first show that there exists a deterministic ABA that can find the outer boundary of P, no matter where in P its initial position may have been. In the following description of this ABA, it is understood that all marks made by it do not interfere with each other or with the original values of the points of P, and that if it passes through a point twice in following a boundary, the marks it makes at that point do not interfere with one another. Informally, our ABA operates as follows:

1) Whenever possible, it moves upward.
2) If it hits a boundary, it marks the point b at which this happened, and begins to follow the boundary. At each step

of the boundary following process, if it has just moved downward, it writes a special mark; in addition, before the first step, it writes one of the special marks at b. If it has just moved upward, it temporarily marks the boundary point b' where this happened, goes back along the boundary, erases the first special mark it finds, returns to b', erases its temporary mark, and continues following the boundary. (Note that this procedure, in effect, uses the boundary as a pushdown stack to count the net number of downward moves.)

3) If it reaches the point b while looking for a special mark to erase, and b no longer has a special mark, b' must be strictly higher than b; it then erases the mark at b and returns to b'. If it is possible to move upward from b', our ABA erases the mark at b' and returns to step (1). If b' has a # above it, our ABA marks it the way b was originally marked and continues as in step (2), treating b' as the new b.

4) If the boundary being followed is the boundary of a hole, the points on it that have #'s above them cannot be its highest points; thus this procedure will eventually get us to a point from which it is possible to return to (1). On the other hand, if we are following the outer boundary, and b has nothing but #'s directly above it, then (2-3) will eventually get b to the topmost row of P, and will follow the outer boundary completely around to b again without ever erasing the special mark at b. Thus if this

ever happens, we know that the boundary on which b lies is the outer boundary of P, and that b is on the top row of P. Moreover, it must happen eventually, since each repetition of (1-3) takes us to a strictly higher point b, and this cannot continue indefinitely in a finite P.

We can next go around the outer boundary again and mark it, so that it can be distinguished from now on from the boundaries of holes. To complete the proof of the theorem, we shall now show how an ABA, beginning at any point c of the outer boundary that has a # to its left, can visit every point of P that lies on that row of the array between c and the first outer boundary point to the right of P. By doing this for every such c, the ABA can visit every point of P, so that in particular it can test whether any point of P has a value in Σ_2.

Starting from c, the ABA moves to the right until it reaches a boundary point. If this point is on the outer boundary, we are done. If it is on the boundary of a hole, we can follow that boundary completely around, mark those points of it that do not have #'s to their right and lie on the same row as c (by keeping a count of upward and downward moves), and then find the leftmost of these points, call it c', by keeping a count of leftward and rightward moves. We now erase all marks, go back to c' and move to the right until we reach a boundary point again; the procedure is then repeated. Since c' is strictly to the right of c, this process must eventually reach the outer boundary.//

The proof of Theorem 4 also shows that an ABA can always find a distinguished point of P, e.g. its upper left-hand corner; there is thus no need to use a distinguished starting point in defining acceptance of an array by an ABA.

4. Machines that cannot rewrite #'s

In Section 3 we defined an ABA as a machine that "bounces off" #'s and does not rewrite them. In this section we prove that the power of such a machine is not increased if we allow it to travel on #'s but not rewrite them. It should be pointed out that the analogous statement about FSAA's is false, as may be seen from the example of the set of arrays whose non-#'s look like

AA...AA
A
AA...AA

(n A's on each "arm", for arbitrary n). Evidently, this set cannot be accepted by an FSAA that cannot traverse #'s, but can be accepted by a deterministic FSAA that is allowed to travel across #'s. Thus being able to move across #'s without rewriting them can increase the power of a machine in the array case, even though it clearly cannot in the string case.

Theorem 5. Let S be the set of input arrays accepted by a TAA that does not rewrite #'s (so that every triple in the image of any ((p,X),#) under δ is of the form ((q,Y),#,Y)). Then there exists an ABA that also accepts just S.

Proof: Let T be the given TAA; our ABA T' will be constructed to act like T whenever T is on a non-#, and to simulate T's position on the #'s while itself remaining on the boundary of the set P of non-#'s. Clearly, we need only show that T' can determine the point of reentry of T into P and its state at that point-- or, if T is nondeterministic, the set of reentry points and associated reentry states. If T never reenters P, either by cycling in a finite region or by going off to "infinity", T'

must also cycle. The following discussion concerns the deterministic case; the extension to nondeterministic TAA's will be described later.

Let $x = (i_0, j_0)$ be the boundary point of P at which T leaves P, and let y be the adjacent point, having value #, onto which T moves. Let E_y be the connected component of #'s that contains y, and let B_x be the set of points of P that are adjacent to E_y; thus B_x is either the outer boundary of P or the boundary of a hole in which y lies. Clearly E_y is finite if and only if y is in a hole.

The point of reentry of T into P must be some point z of B_x. We shall show that T' can keep track of T's position in E_y relative to x by recording two integers, the horizontal and vertical displacements of T. Once T' has calculated these displacements, it can determine whether any point of B_x has these coordinates and can go there. (T' can also keep track of T's current state within its own state set, since T has only finitely many states.) If there is no such point of B_x, so that T is still in E_y, T' continues its simulation.

We must now show that if T reenters P, T' always has room to record T's coordinates. In fact, we shall prove that T must reenter P within $|Q||B_x|$ moves, where $|Q|$ is the number of states of T and $|B_x|$ is the length of B_x, if it ever reenters at all. Thus the maximum displacement of T from x before reentry is $|Q||B_x|$, so that the points of B_x can be used to record the displacements using base $|Q|$ representation.

Note first that when a TAA that cannot write on #'s moves onto them, it behaves like an FSAA with constant input. By the

motion of an FSAA on an array will be meant the word $\omega \in \Delta^*$ that gives the sequence of directions of its moves on the array. For example, the motion corresponding to a raster scan of an m by n rectangle is $R^nDL^nD...DL^n$ (if m is even; $...DR^n$, if m is odd), where there are m-1 D's.

Lemma. The motion of a deterministic FSAA on an infinite array of #'s is eventually periodic, i.e. $\omega = \alpha\beta^*$, where α, β are finite strings over Δ. (An analogous result holds for non-writing array automata that have pushdown storage.)

Proof: Let M be the set of states of the FSAA; since it has constant input, it must begin to cycle within $|M|$ moves; thus its state sequence is of the form $\pi\rho^*$, where π, ρ are finite strings in M^*. Since the FSAA is deterministic, we have a well-defined mapping which associates with each state $s \in M$ the direction in which the FSAA moves upon entering s; under this mapping, $\pi\rho^*$ goes into some $\alpha\beta^*$ of the desired type. Moreover, $|\beta| \leq |M|$. //

Let $|\omega|_a$ be the number of occurrences of the symbol a in the string ω. If $\alpha\beta^*$ is as in the Lemma, and $|\beta|_L = |\beta|_R$, $|\beta|_D = |\beta|_U$, the FSAA cycles within a finite region without ever leaving it. Otherwise, the FSAA moves off to "infinity" in the direction of the excesses, each cycle increasing its displacements from its starting point by the exact amounts of the excesses; thus its motion is "linear". (It follows that a deterministic FSAA can never visit every point in an infinite array of #'s.)

Let $\alpha\beta^* \in \Delta^*$ be T's motion on E_y. We make two assumptions:

1) α is null. This is no restriction, since α can always be chosen so that $|\alpha| \leq |Q|$, and T' can simulate the first $|Q|$ moves of T within its own memory.

2) β is defined so that if T reenters P, it does so at the end of a β cycle; this will simplify the argument below.

We can now show that T reenters P within $|Q||B_x|$ moves, if at all. Suppose T reenters P at the end of the kth β cycle. Let $i = |\beta|_R - |\beta|_L$, $j = |\beta|_U - |\beta|_D$. Then the position of the reentry point is $z = (i_0, j_0) + k(i,j)$. Thus the city block distance from x to z is $|z-x| = k|i|+k|j| = k(|i|+|j|)$. But $|i|+|j| \geq 1$, so $k \leq |z-x| \leq \max_{z',x' \in B_x} |z'-x'|$ = diameter of $B_x \leq |B_x|$. Since $|\beta| \leq |Q|$, it follows that $k|\beta| \leq |Q||B_x|$. But $k|\beta|$ is the number of moves T makes on E_y before reentering P; we have thus shown that if T reenters P, it does so within $|Q||B_x|$ moves. If $k|\beta| > |Q||B_x|$, T can no longer reenter P, either because it is already so far away that it can never get back to B_x on its "linear" path to "infinity", or because it is cycling on a closed path that does not meet B_x.

In summary: Whenever T has moved onto a #, say at y, T' marks its point x of departure, and the border B_x that contains x, and stores T's exit state in its memory. It then uses B_x to compute T's position in E_y at each successive move. T then visits every point of B_x in turn and decides whether that point's displacement from x is the same as the computed position. If so, it erases the temporary information stored on B_x, goes to the reentry point, and enters the proper state; if not, it continues the simulation.

If T is nondeterministic, the description of T' is analogous. T's motion on E_y is now described by a disjunction of $\alpha\beta^*$'s, and there may be many reentry points. Moreover, since T need not be

moving "linearly", it may take a path that carries it very far from P and yet brings it back to P; thus there will be no room on B_x to record all the displacements on any reentrant path. However, it is easily seen that in any such case, T has a shorter path which allows it to reenter P without overflowing the capacity of B_x. Thus when T' nondeterministically simulates T, it too has the possibility of choosing this shorter path. //

References

[1] M. Blum and C. Hewitt, Automata on a 2-dimensional tape, IEEE Symp. on Switching and Automata Theory, 1967, 155-160.

[2] M. J. Fischer, Two characterizations of the context-sensitive languages, ibid., 1969, 149-156.

[3] J. P. Mylopoulos, On the definition and recognition of patterns in discrete spaces, Technical Report 84, Computer Sciences Laboratory, Department of Electrical Engineering, Princeton University, August 1970.

[4] R. A. Kirsch, Computer interpretation of English text and picture patterns, Trans. IEEE EC-13, 1964, 363-376.

[5] A. Rosenfeld, "Isotonic grammars, parallel grammars, and picture grammars", in D. Michie and B. Meltzer, eds., Machine Intelligence VI, University of Edinburgh Press, 1970, in press.